INTRODUCTION

For this project, I will implement an interpreter for LOLCODE, an esoteric programming language inspired by LOLCats. The description of LOLCODE can be found in the third section.

DESCRIPTION OF APPROACH

The interpreter for LOLCODE will be written in Python3, taking advantage of the various features of Python3 to speed up the evaluation process. The interpreter will parse LOLCODE on-the-fly, reading and evaluating each token as it is encountered. Variables will be stored using Python3's built-in `dict` type, which provides mappings between key-value pairs. Each key or value may be an n-tuple or any other built-in type in Python3. This allows for the interpreter to keep track of all variables in a single `dict`. Evaluation will be performed with the Python3 equivalent of the operation that is performed in LOLCODE.

LANGUAGE IMPLEMENTED

This section contains the syntax rules and specifications for LOLCODE.

1. Formatting
    1. Each LOLCODE program must begin with the keyword `HAI` and end with the keyword `KTHXBAI`. Anything before or after these keywords will be ignored.
    2. Whitespace
        1. Spaces will be used to separate tokens in the language, although some tokens contain spaces in them.

2.Multiple spaces and tabs are treated as a single whitespace token; indentation is irrelevant.

3.A command starts at the beginning of a line and ends with the newline character. Multiple commands may be placed on the same line if they are separated by a comma (`,`). In this case, the comma acts as a virtual newline or a soft-command-break.

4.Soft-command-breaks are ignored inside quoted strings. An unterminated string literal will cause an error.

3.Comments

1.Single line comments are begun with `BTW` and may occur either on a separate line or after a line of code following a line separator (,). The following are all valid single line comments:

```
I HAS A VAR ITZ 12,      BTW VAR = 12
I HAS A VAR ITZ 12
BTW VAR = 12
```

2.Multiline comments are begun with `OBTW` and ended with `TLDR`. They should be started on their own line, or following a line of code following a line separator. These are valid multiline comments:

```
I HAS A VAR ITZ 12
     OBTW this is a long comment block
          see, i have more comments here
          and here
     TLDR
I HAS A FISH ITZ BOB

I HAS A VAR ITZ 12, OBTW this is a long comment
                    block see, i have more
                    comments here and here
TLDR, I HAS A FISH ITZ BOB
```

2.Variables

1.Scope - There is only a global scope.

2.Naming

1.Variable identifiers must begin with a letter and must be a combination of letters, numbers, and underscores. No other symbols are allowed.

2.Variable identifiers are case sensitive. “cheezburger”, “CHEEZBURGER”, and “CheezBurger” would all be different identifiers.

3.Declaration and assignment

1.To declare a variable the keyword `I HAS A` is followed by the variable name. To assign the variable a value within the same statement, follow the variable name with `ITZ <value>`.

2.Assignment of a variable is accomplished with an assignment statement, `<variable> R <expression>`.

3.The following are all valid variable declarations:

```
I HAS A VAR     BTW VAR is null and untyped
VAR R “three”   OBTW VAR is now a YARN and equals
                “three” TLDR
VAR R 3         OBTW VAR is now a NUMBR and equals
                3 TLDR
```

3.Primitive Types

1.The primitive types that LOLCODE recognizes are:

1.strings (YARN)

1.String literals (YARN) are marked with double quotation marks (“).

2.Line continuation & soft-command-breaks are ignored inside quoted strings.

3.An unterminated string literal (no closing quote) will cause an error.

2.integers (NUMBR)

1.A NUMBR is an integer as specified in the host implementation/architecture.

2.Any contiguous sequence of digits outside of a quoted YARN and not containing a decimal point (.) is considered a NUMBR.

3.A NUMBR may have a leading hyphen (-) to signify a negative number.

3.floats (NUMBAR)

1.A NUMBAR is a float as specified in the host implementation/architecture.

2.It is represented as a contiguous string of digits containing exactly one decimal point.

3.Casting a NUMBAR to a NUMBR truncates the decimal portion of the floating point number.

4.A NUMBAR may have a leading hyphen (-) to signify a negative number.

4.booleans (TROOF)

1.The two boolean (TROOF) values are WIN (true) and FAIL (false).

2.The empty string ("") and numerical zero are all cast to FAIL. All other values evaluate to WIN.

5.untyped (NOOB)

1.The untyped type (NOOB) cannot be cast into any type except a TROOF. A cast into TROOF makes the variable FAIL.

2.Any operations on a NOOB that assume another type (e.g., math) results in an error.

3.Explicit casts of a NOOB (untyped, uninitialized) to any other type results in an error.

2.Typing is handled dynamically and until a variable is given an initial value, it is untyped (NOOB).

3.IT

1.A bare expression (e.g. a function call or math operation), without any assignment, is a legal statement in LOLCODE.

2.Aside from any side-effects from the expression when evaluated, the final value is placed in the variable IT.

3.IT's value remains in local scope and exists until the next time it is replaced with a bare expression.

4.Operators

1.Calling syntax and precedence

1.Mathematical operators and functions rely on prefix notation. By doing this, it is possible to call and compose operations with a minimum of explicit grouping. When all operators and functions have known arity, no grouping markers are necessary. In cases where operators have variable arity, the operation is closed with `MKAY?`.

2.Calling unary operators then has the following syntax:

```
<operator> <expression1>
```

3.The AN keyword can optionally be used to separate arguments, so a binary operator expression has the following syntax:

```
<operator> <expression1> [AN] <expression2>
```

4.An expression containing an operator with infinite arity can then be expressed with the following syntax:

```
<operator> <expr1>[[[AN] <expr2>] [AN] <expr3>…] MKAY?
```

2.Math

1.The basic math operations are binary prefix operators:

```
SUM OF <x> AN <y>,        BTW  +
DIFF OF <x> AN <y>,       BTW  -
PRODUKT OF <x> AN <y>,    BTW  *
QUOSHUNT OF <x> AN <y>,   BTW  /
MOD OF <x> AN <y>,        BTW  modulo
BIGGR OF <x> AN <y>,      BTW  max
SMALLR OF <x> AN <y>,     BTW  min
```

2.`<x>` and `<y>` may each be expressions in the above, so mathematical operators can be nested and grouped indefinitely.

3.Math is always performed as floating point math regardless of whether the expressions are NUMBRs or NUMBARs.

4.If one or another of the arguments cannot be safely cast to a numerical type, then it fails with an error.

3.Boolean - Boolean operators working on TROOFs are as follows:

```
BOTH OF <x> [AN] <y>,     BTW  and: WIN iff x=WIN, y=WIN
EITHER OF <x> [AN] <y>,   BTW  or: FAIL iff x=FAIL, y=FAIL
WON OF <x> [AN] <y>,      BTW  xor: FAIL if x=y
NOT <x>,                  BTW unary negation: WIN if x=FAIL
ALL OF <x> [AN] <y> … MKAY?,  BTW infinite arity AND
ANY OF <x> [AN] <y> … MKAY?,  BTW infinite arity OR
```

4.Comparisons

1.Comparison is done with two binary equality operators:

```
BOTH SAEM <x> [AN] <y>,    BTW  WIN iff x == y
DIFFRINT <x> [AN] <y>,     BTW  WIN iff x != y
```

2.Comparisons are performed as integer math in the presence of two NUMBRs, but if either of the expressions are NUMBARs, then floating point math takes over. Otherwise, there is no automatic casting in the equality, so `BOTH SAEM "3" AN 3` is `FAIL`.

3.There are no special numerical comparison operators. Greater-than and similar comparisons are done idiomatically using the minimum and maximum operators.

```
BOTH SAEM <x> AN BIGGR OF <x> AN <y>   BTW x >= y
BOTH SAEM <x> AN SMALLR OF <x> AN <y>  BTW x <= y
DIFFRINT <x> AN SMALLR OF <x> AN <y>   BTW x > y
DIFFRINT <x> AN BIGGR OF <x> AN <y>    BTW x < y
```

4.If <x> in the above formulations is too verbose or difficult to compute, the automatically created IT variable comes in handy. A further idiom could then be:

```
<expression>, DIFFRINT IT AN SMALLR OF IT AN <y>
```

5.Casting

1.Operators that work on specific types implicitly cast parameter values of other types. If the value cannot be safely cast, then it results in an error.

2.An expression's value may be explicitly cast with the binary `MAEK` operator.

```
MAEK <expression> [A] <type>
```

3.Where `<type>` is one of `TROOF`, `YARN`, `NUMBR`, or `NUMBAR`. This is only for local casting i.e.,only the resultant value is cast, not the underlying variable(s), if any.

4.To explicitly re-cast a variable, you may create a normal assignment statement with the `MAEK` operator:

```
<variable> R MAEK <variable> [A] <type>
```

5.Input/Output

1.Input and output in LOLCODE is terminal-based.

2.The print (to STDOUT or the terminal) operator is `VISIBLE`. It has infinite arity and implicitly concatenates all of its arguments after casting them to `YARNs`. It is terminated by the keyword `MKAY?`, followed by the statement delimiter (line end or comma) and the output is automatically terminated with a carriage return.

```
VISIBLE <expression> [<expression> …] MKAY?
```

3.To accept input from the user, the keyword is

```
GIMMEH <variable>
```

which implicitly tries to cast the input as a NUMBR, NUMBAR, TROOF, or NOOB. If the implicit cast fails then the input is assumed to be YARN and the resultant value is stored in the given variable.

6.Flow Control

1.The traditional if-then clause takes advantage of the implicit IT variable and has the form:

```
<expression>
O RLY?
     YA RLY
          <code block>
     [MEBBE <expression>
          <code block>
     [MEBBE <expression>
          <code block>
     …]]
     [NO WAI
          <code block>]
OIC
```

1.O RLY? branches to the block begun with YA RLY if IT can be cast to WIN, and branches to the NO WAI block if IT is FAIL. The code block introduced with YA RLY is implicitly closed when NO WAI is reached. The NO WAI block is closed with OIC.

2.Optional MEBBE <expression> blocks may appear between the YA RLY and NO WAI blocks. If the <expression> following MEBBE is true, then that block is performed; if not, the block is skipped until the following MEBBE, NO WAI, or OIC.

3.An example of this conditional is:

```
BOTH SAEM ANIMAL AN "CAT"
O RLY?
YA RLY, VISIBLE "J00 HAV A CAT"
MEBBE BOTH SAEM ANIMAL AN "MAUS"
     VISIBLE "NOM NOM NOM. I EATED IT."
NO WAI, VISIBLE “JOO SUX”
OIC
```

2.Loops

1.Loops are demarcated with `IM IN YR <label>` and `IM OUTTA YR <label>`.

2.Iteration loops have the form:

```
IM IN YR <label> WILE <expression>
     <code block>
IM OUTTA YR <label>
```

3.The `WILE <expression>` evaluates the expression as a `TROOF`: if it evaluates as `WIN`, the loop continues once more, if not, then loop execution stops, and continues after the matching `IM OUTTA YR <label>`.

7.Functions

EXAMPLE PROGRAMS

Hello World.lol

```
HAI            BTW  Hello World
VISIBLE “HAI WORLD!!!!”
KTHXBAI
```

Count.lol

```
HAI
I HAS A VAR ITZ 0
```

```
BTW for(; VAR <= 10; VAR++)
IM IN YR LOOP UPPIN YR VAR TIL DIFFRINT VAR AN SMALLR OF VAR AN
10
     VISIBLE VAR
IM OUTTA YR LOOP
KTHXBAI
```

Fibonacci Numbers.lol

```
HAI       BTW Fibonacci Numbers
I HAS A MAX
I HAS A COUNT ITZ 0
I HAS A NUM ITZ 1
I HAS A NEXT ITZ 1

VISIBLE "How many fibonacci numbers do you wish to see?"
GIMMEH MAX

DIFFRINT COUNT AND BIGGR OF COUNT AN MAX     BTW if COUNT < MAX
O RLY?
    YA RLY
        VISIBLE NEXT
        COUNT R SUM OF COUNT AN 1       BTW COUNT++
OIC

DIFFRINT COUNT AND BIGGR OF COUNT AN MAX     BTW if COUNT < MAX
O RLY?
    YA RLY
        VISIBLE NEXT
        COUNT R SUM OF COUNT AN 1       BTW COUNT++
        NEXT R SUM OF NEXT AN NUM       BTW NEXT = NEXT + NUM
OIC

BTW for(; COUNT < MAX; COUNT++)
IM IN YR loop UPPIN YR COUNT WILE DIFFRINT COUNT AN BIGGR OF
COUNT AN MAX
    VISIBLE NEXT
    SUM OF NEXT AN NUM        BTW IT = NEXT + NUM
    NUM R NEXT                BTW NUM = NEXT
    NEXT R IT                 BTW NEXT = IT
IM OUTTA YR loop

KTHXBAI
```